



SPIN WHEEL EFFECT

by 3Point Games

USER MANUAL

v1.6

1. Overview

The Spin Wheel Effect (**SWE**) is a C# plugin for Unity3D thought to help developers to avoid the ugly **Stroboscopic Effect** (a.k.a. **Wagon Wheel Effect**), which occurs when a wheel rotates faster than the frame rate of the game.

The wagon-wheel effect is an optical illusion in which a spoked wheel appears to rotate differently from its true rotation. The wheel can appear to rotate more slowly than the true rotation, it can appear stationary, or it can appear to rotate in the opposite direction from the true rotation.

[Source: Wikipedia]

You can consider this system as a **fake motion blur**, since it gives the same feels of a motion blur but it only simulates it, dramatically increasing the runtime performance.

2. How to setup the system

In order to correctly set up the system there are some preliminary steps the user must follow.

2.1 Create well-structured models

This step is the most important one and failing in this could produce unwanted behaviors.

The automatic texture generation process for a wheel needs a well-structured model (UV Maps, too!), since it works only with the model's meshes and

materials. If you already have some wheel models, you may have to split them in some parts, following this scheme:

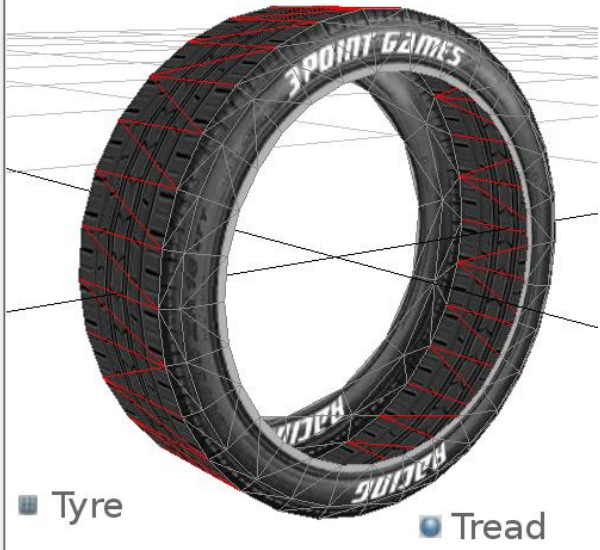
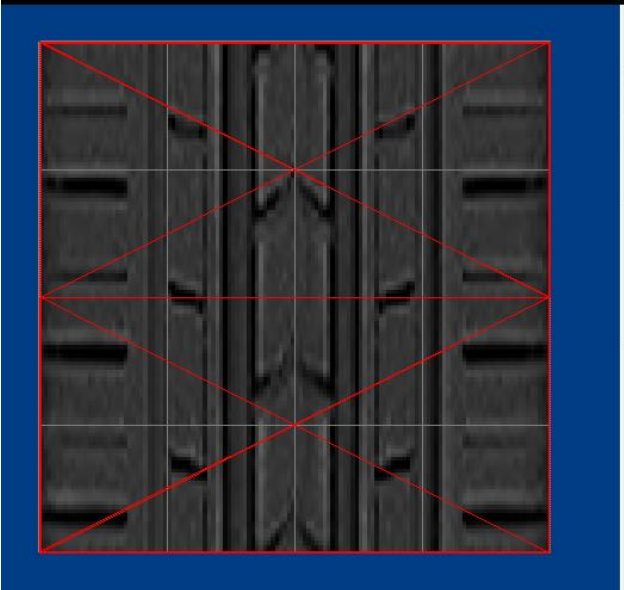
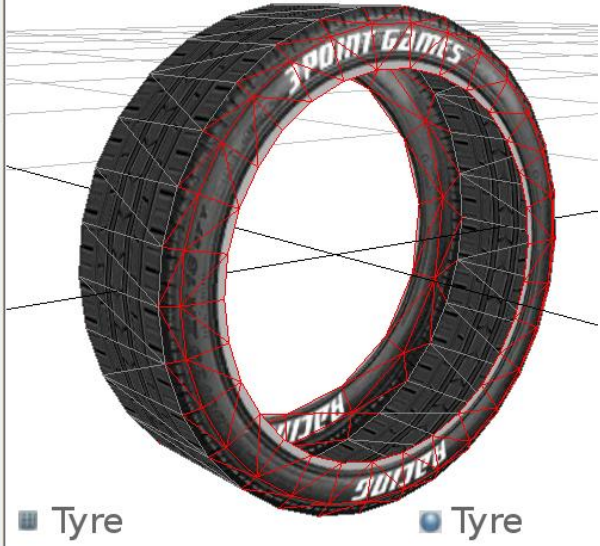
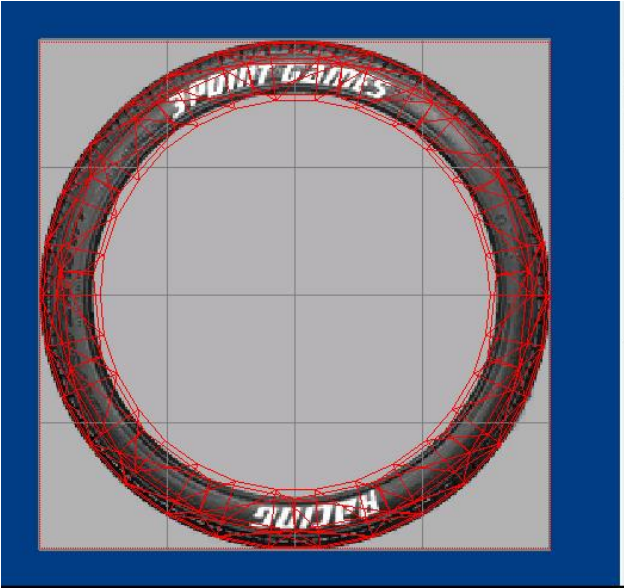
Mesh	Material(s)
■ Tyre	● Tyre ● Tread
■ Rim	● Rim
■ Spokes	● Rim ● Details
■ SpokesSpin	● SpokesSpin
■ RimCenter	● Rim
■ BrakeDisk	● BrakeDisk

2.1.1 TYRE

A SWE ■ Tyre model is an only mesh with two materials: ● Tyre and ● Tread.

- Tyre's UV Map must be **front planar**.
- Tread's UV Map must be **rectangular overlapped**.

This difference allows the system to blur the textures in two different ways: the tyre's ones with a radial blur and the tread's ones with a directional (vertical) blur.

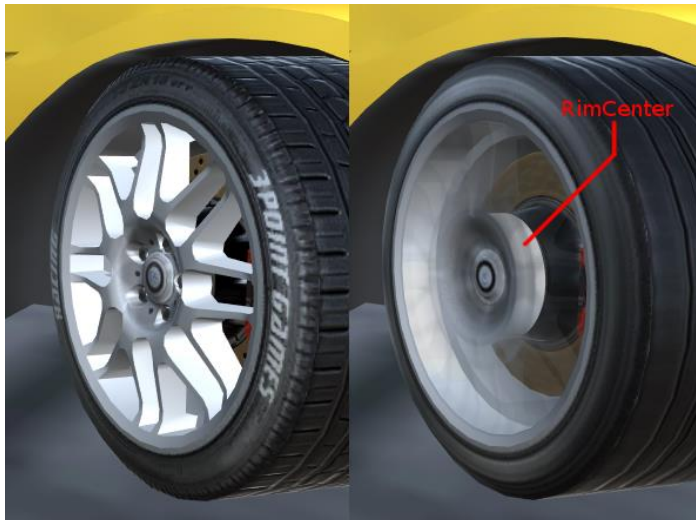


2.1.2 RIM, SPOKES & RIM CENTER

- The **Rim** mesh is the cylindrical part of the rim, without the spokes. It has an only material, conventionally named **Rim**
- The **Spokes** mesh is, obviously, the 3D representation of the spokes. It uses two materials:
 - **Rim**, the same of the other rim's parts
 - **Details**, distinguish logos and bolts from the spokes

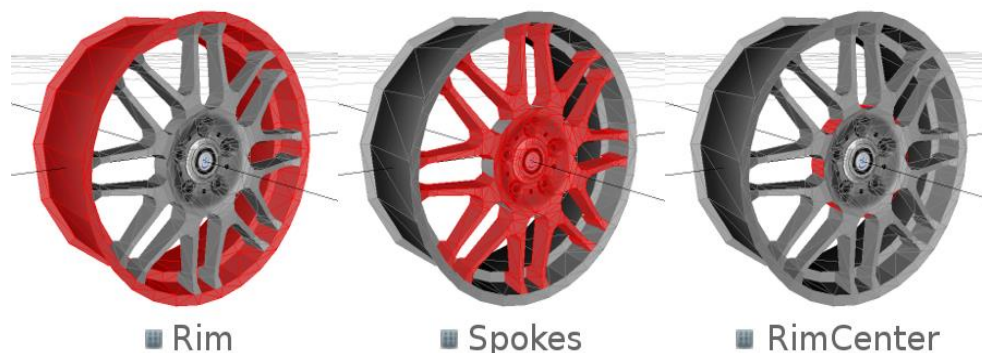
Please note that the top spoke must be vertical

- The **RimCenter** mesh is something that is not visible when the wheel is not moving but only when the effect is running



As you can see, once the effect blurs the spokes, a little cylinder appears in the middle of the wheel, giving a quite realistic feeling.

*It represents an always fully opaque area that is located at the middle of the rim. It uses the **Rim** material and the system fades it in/out automatically.*

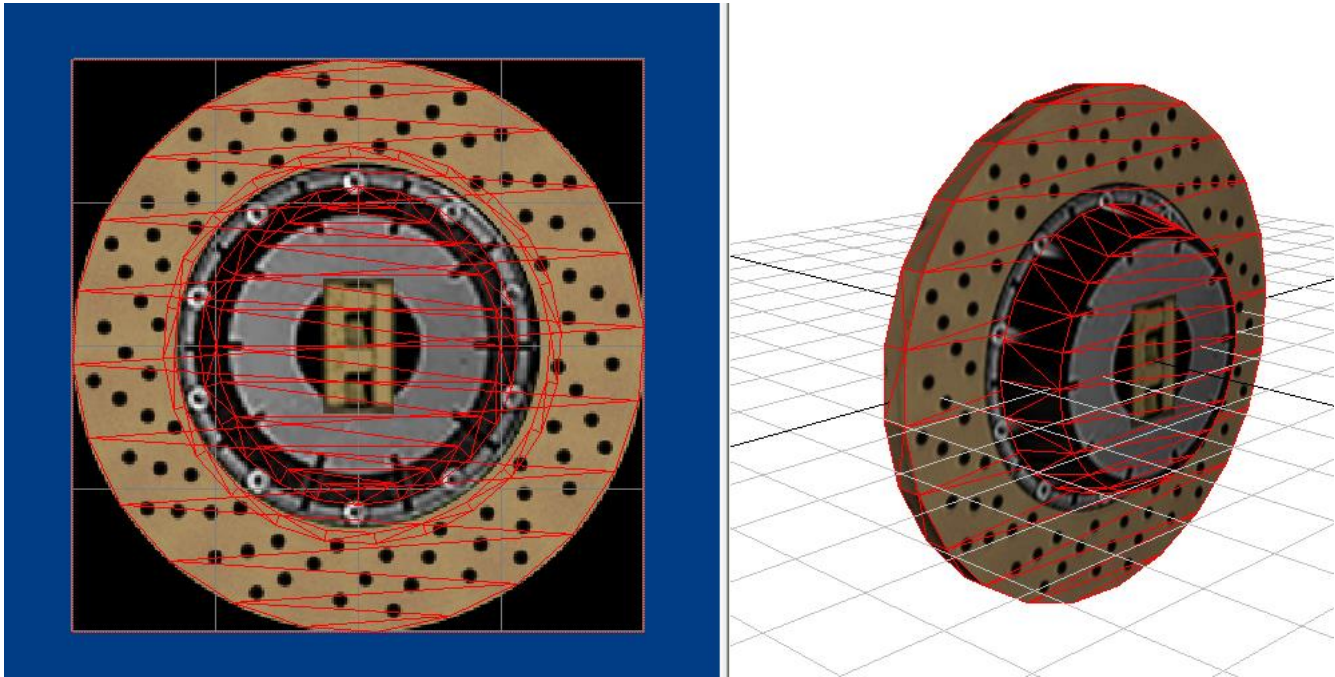


The **Rim**'s faces must be UV Mapped with a planar frontal projection, while the **RimCenter**'s ones doesn't need a fixed UV Map, so make it as you prefer. Here is an example:



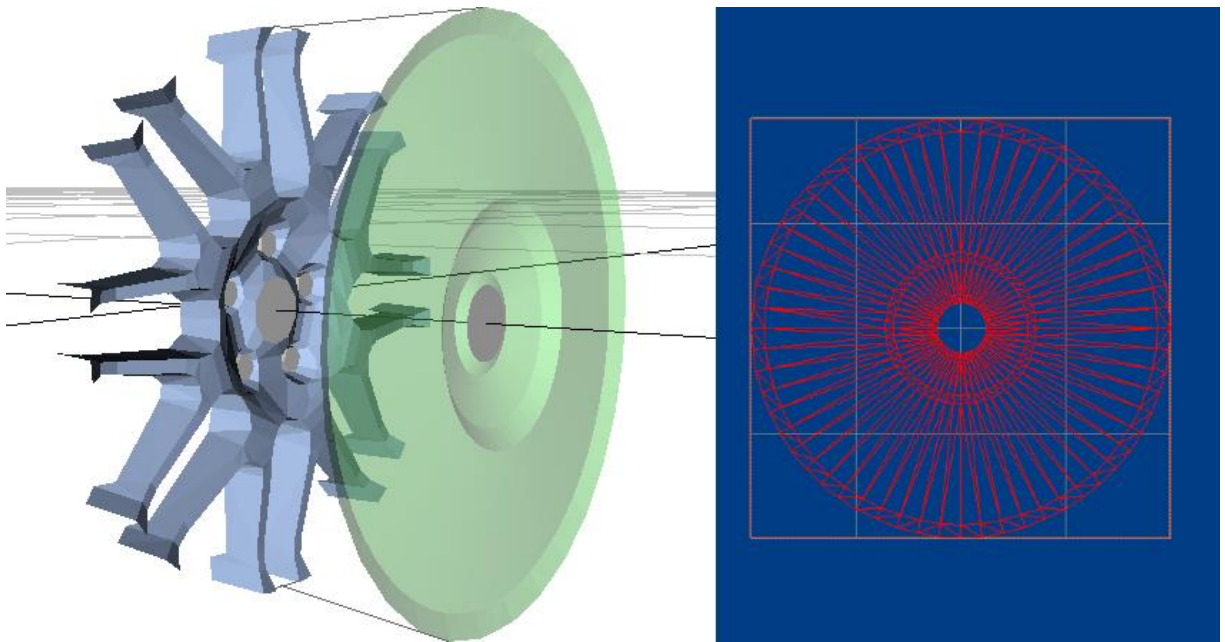
2.1.3 BRAKE DISK

A SWE **BrakeDisk** model just needs the **BrakeDisk** material with a front planar UV Map, eventually relaxing the side faces:



2.1.4 SPOKES SPIN

The SWE **SpokesSpin** model is the most important one and the minimal part of the effect. It is the representation of the blurred spokes during the wheel rotation. You can think to this mesh as a “coverage” of the spokes mesh (like putting a sheet over it). It will be faded-in while the **Spokes** will be faded-out.








The screenshot shows a detached mesh for **SpokesSpin**, but it should be very close to the **Spokes** mesh (not completely welded because of some well-known depth rendering issues).

Future optimizations may simplify the modeling & texturing workflow, reduce the number of the needed materials and/or the needed meshes; this will be one of the greatest objectives for new versions of SWE.

2.2 Texturing

This system currently supports only **Diffuse** and **Bump** maps for both PC and Mobile platforms.

You can use bump maps on each material, and SWE will blur the maps of the following materials:

<i>Material</i>	<i>Supported maps</i>	<i>Blur type</i>
 Tyre	✓ Diffuse ✓ Bump	Radial
 Tread	✓ Diffuse ✓ Bump	Directional
 Rim	✓ Diffuse	Radial
 SpokesSpin	✓ Diffuse	Radial
 BrakeDisk	✓ Diffuse ✓ Bump	Radial

S.W.E. will use all the textures in the materials, if supported. This means that if you do not want to use some diffuse or bump maps somewhere, simply don't use them in the material inspector.

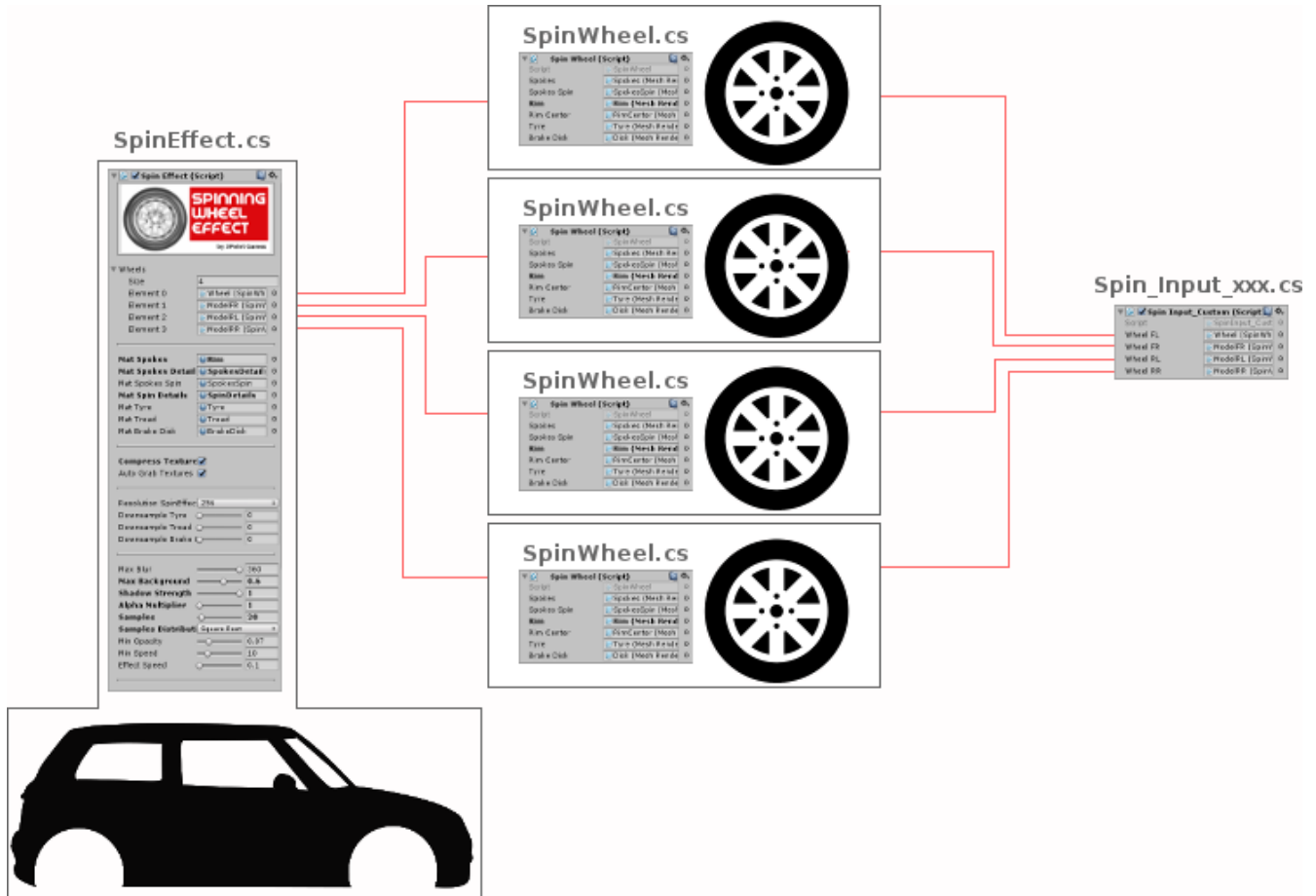
In most car games, the scene's geometry often hides the wheels and depending on screen and camera settings, they can appear far and small. In this case, bump maps are useless.

Note: S.W.E. works with most default shaders. It's important that they use the following names: “_MainTex” for diffuse and “_BumpMap” for bump map.

2.3 Configuring S.W.E.

3.3.1. Scripts architecture

S.W.E. uses a simple script architecture, which follows by this scheme:



As convention, I recommend to attach the *SpinEffect.cs* script to the root object of the car, the *SpinWheel.cs* scripts directly to the wheels and the *SpinInput_xxx.cs* script on some other object (the car itself it's a good choice). This will help to get support.

3.3.2 Scripts overview

SpinEffect.cs

This is the main script of S.W.E.

It stores information about wheels, materials and settings to generate the textures and swap them at runtime. Here is how the inspector will look like:

Shading mode: *If you use the Standard Shader for ALL the wheel materials it's recommended to set the Standard mode, but if you use ALL legacy shaders you must use the Legacy mode. Using both Standard and Legacy shaders in the same wheel may work but it's not guaranteed.*

Wheels: *all the wheels to be blurred while spinning (they must have a SpinWheel component)*

Spokes No: *the number of the spokes of the wheel(s) used in this configuration of S.W.E. Using a wrong number or 0 causes bad positioning of the spin texture.*

Materials: *all the materials the system will use. These materials will be assigned to the right meshes, so it would be better to put in the list the materials already used in the scene's meshes*

Compress textures: *if on, it compresses the generated textures in the High Quality DXT format. Useful to reduce the memory usage (only for DXT supporting platforms). Note that this will slow the generation process, since the DXT compression needs a Texture2D, and the conversion from RenderTexture to Texture2D is quite slow. For a total of 100 textures with size 256x256 the process will take about 0.1s if they are uncompressed and about 2s if they are compressed (200 times slower). [recommended: OFF]*

Auto grab textures: *if on, the system will use as reference textures the ones already set in the chosen materials [recommended: ON]*

Resolution Spin Effect: *the resolution of the generated textures to be applied to the SpinRim mesh.*

Rim downsample: *the downsample factor of the rim's generated texture*

Tyre downsample: *the downsample factor of the tyre's generated texture*

Tread downsample: *the downsample factor of the tread's generated texture*

BrakeDisk downsample: *the downsample factor of the brake disk's generated texture*

Max blur: the maximum angle of the spin blur effect. Generally, the maximum blur should be 360, but something like 90-100 will be good, too. Use lower values to limit the blurriness at high speeds

Alpha multiplier: multiplication factor for the generated transparent maps. Generally, a value between 1 and 2 will be good

Samples: how many textures the system have to generate for each map? 10-20 is a good value

Samples distribution: the distribution of the computed blur angle over the total rotation space $[0^\circ, 360^\circ]$. It uses different mathematic formulas. Useful for a better low speed blur spreading when using a small amount of textures

Min opacity: if this variable isn't 0, the rim model won't be completely faded out and its alpha will be at least equal to this value

Min Speed: the minimum speed to reach in order to start blurring the wheel

Max Speed: after this speed the wheel will always have the maximum blur

Adaptive occlusion (see 3.4): If true, this adds extra alpha to the spinning mesh accordingly to the angle between the wheel and the camera's view. This adds some realism to the effect, since it simulates the occlusion the side faces of the spokes create on the inner parts of the wheel when you see the wheel from the side. Adaptive occlusion doesn't work on double-sided spokes, i.e. plane propeller. Use it on car wheels only!

Max adaptive occlusion : maximum obscurance of the inner part of the wheel. A value of 1 will completely hide the inner parts at low camera angles, a value of 0 won't obscure anything.

Texture arrays: these arrays are shown in the inspector in play mode as read only. This is intentional, because these arrays are automatically allocated, filled and assigned to the game elements

SpinWheel.cs

For each wheel, you must attach this script to the wheel object and set all the asked meshes. You will need to drag the wheel objects in the SpinEffect.sc script and, eventually, in the input script.

*S.W.E. will use as wheel speed the value passed with the SpinWheel's public function **SetSpeed()**.*

SpinInput_xxx.cs – input handlers

S.W.E. does not know what logic the user uses to move the wheels, so it requires accessing the wheels' speed from the outside. An Input Handler script adapts S.W.E. to the wheel moving system.

*A simple script named **SpinInput_Custom.cs** allows the user to accelerate and brake basing on keyboard or touch input. It's used in the **Complete Car example scene** and it is a good start for writing a custom input handler.*

Input Handlers must always set the wheels' speed by calling the function `SpinWheel.SetSpeed()`.

Since most car games use a physics engine, a custom `SpinInput_xxx.cs` script should access that physics engine's properties and set the SpinWheel's speed accordingly to the engine's wheel speed values.

*S.W.E. is compatible with **UnityCar** and the Input Handler **SpinInput_UnityCar.cs** will link S.W.E. and UnityCar.*

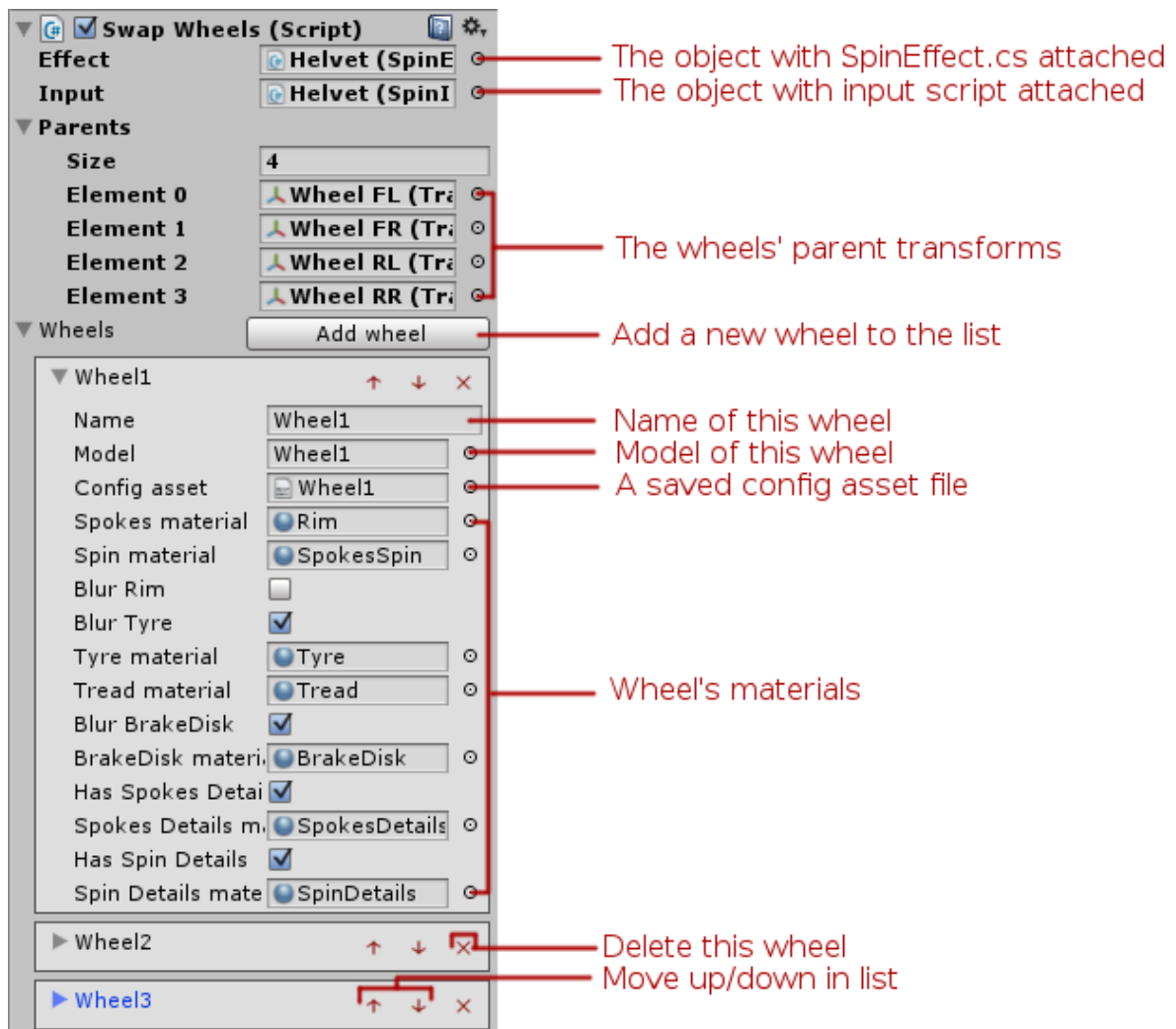
Note: You will need to un-comment the UnityCar input handler if you want to use it.

SwapWheels.cs

This script simulates a real game situation: dynamic wheel spawn/swap.
It can avoid the following questions:

- 1) What should I do when I spawn a wheel at runtime?
- 2) Which values do I have to set in SpinEffect.cs?

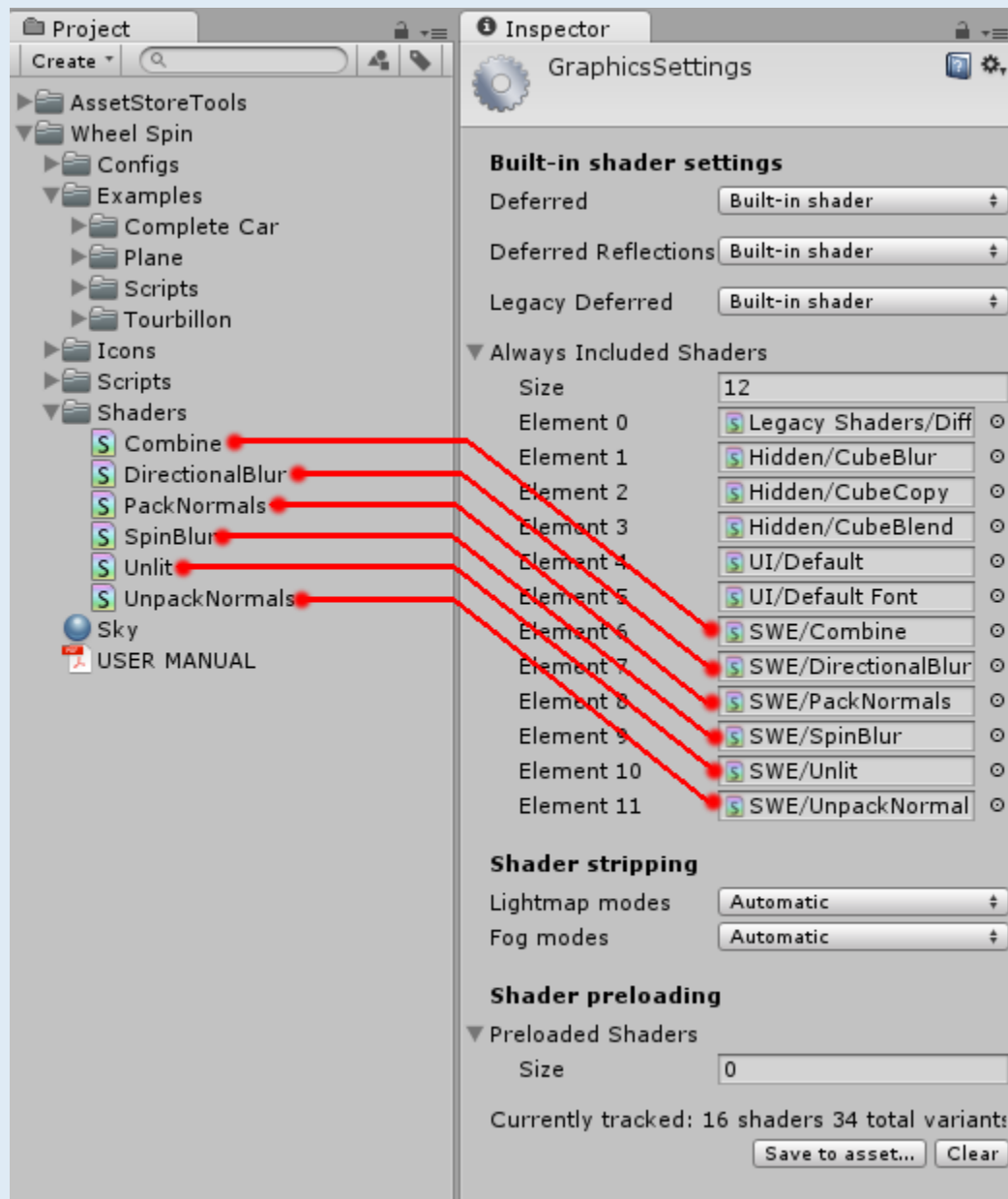
Simply, fill the required fields and add some wheels (for each wheel you should choose some config file). See the example scene named "Scene_Car".



3.3 Build

S.W.E. has been tested on PC and Android, so it should compile without problems at least on these two platforms (any feedback for other platforms build is welcome).

*Anyway there's an important step to do before building your project: add all the S.W.E. shaders in **Edit->Project Settings->Graphics->Always included shaders***



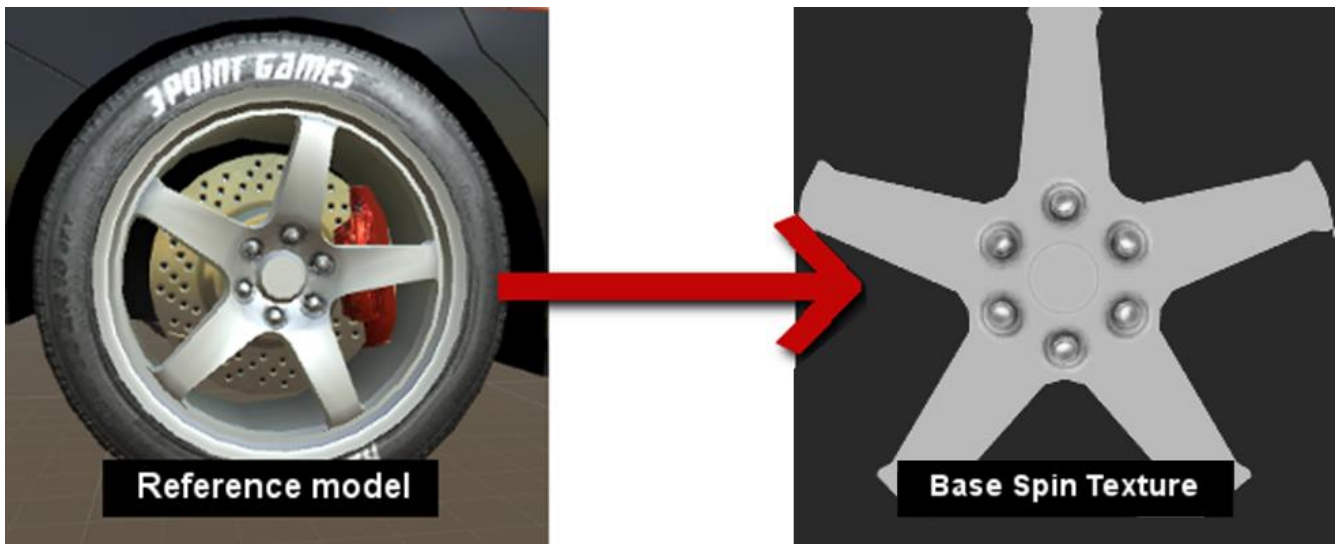
3. Under the hood

SWE generates a **pool of blurred textures** basing on the models' original shape, color and textures and **swaps them at runtime accordingly to the speed of the wheels**. This is completely automatic and the only thing to do is configuring the system. Almost all the texture generation is **GPU-powered**, so SWE can generate dozens of textures in no time! There is no FPS impact at all except for the generation routine that generally operates in less than 0.1 seconds (more than 100x times faster than v1.3!) and in most cases it will be called only one time (at startup). For this reason, the "critical resource" is not the time but the memory usage. The average amount of used memory is 20-50 MB.

But let's look inside this system a bit deeper...

3.1 Base Spin Texture generation

This is the first automatic step S.W.E. takes in order to create all the other textures. A Base Spin Texture is just a front photo of the Spokes mesh of a wheel with an unlit material.



3.2 Blurred Spin Textures generation

S.W.E. uses the Base Spin Texture to generate the Blurred Spin Textures that will be applied on the ■SpokesSpin mesh.

*The system will generate N blurred textures following a D distribution function, where N is the value of the property **samples** and D is the value of **samplesDistribution**. In this way, you can get different blur angles for each texture. Let's make some examples:*

***samples** = 10
samplesDistribution = Linear
maxBlur = 360
textures: | 0° | 36° | 72° | 108° | 144° | 180° | 216° | 252° | 288° | 324° | 360° |*

***samples** = 10
samplesDistribution = SquareRoot
maxBlur = 360
textures: | 0° | 19° | 39° | 59° | 82° | 106° | 133° | 163° | 200° | 247° | 360° |*

***samples** = 10
samplesDistribution = CubeRoot
maxBlur = 360
textures: | 0° | 13° | 26° | 41° | 57° | 75° | 95° | 120° | 150° | 193° | 360° |*

As you can see the CubeRoot distribution have a greater left density. Trying different combinations of distributions is a good approach to get the desired results.

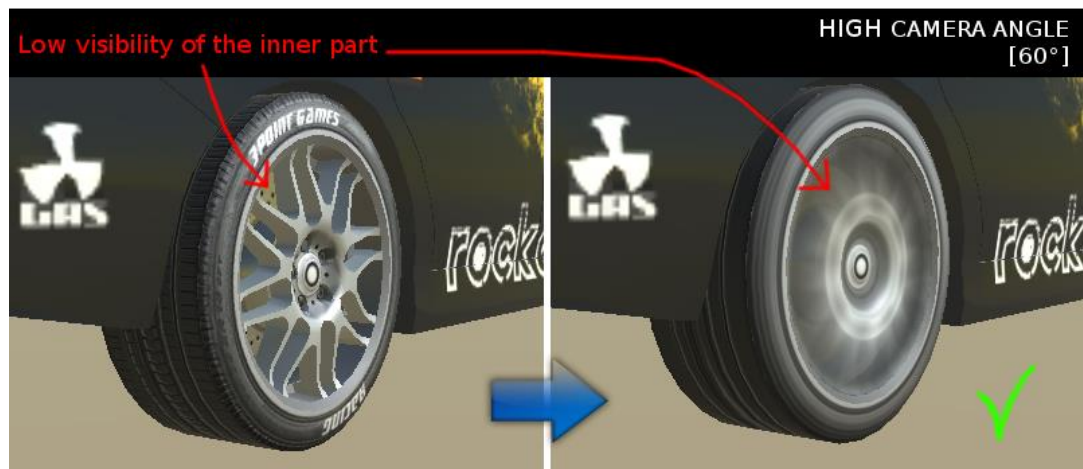
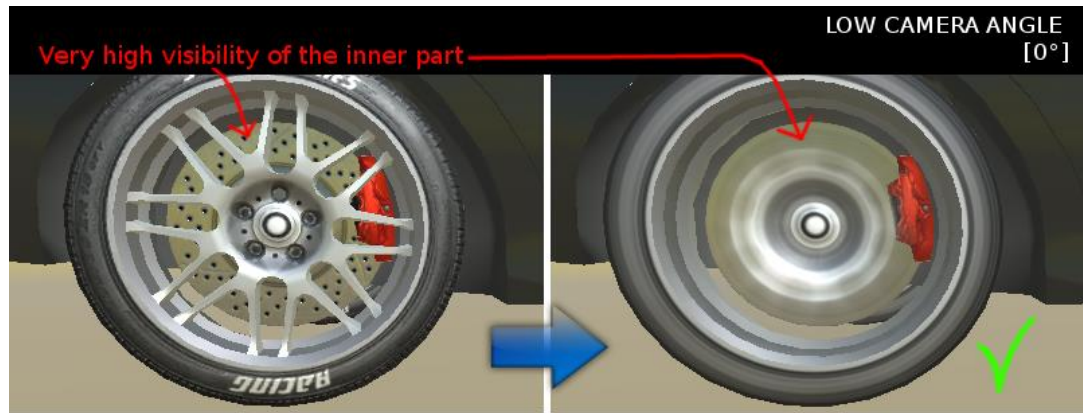
3.3 Runtime Textures Swap

*Once all the textures have been created, the system will automatically swap them accordingly to the parameters **minSpeed** and **maxSpeed**. The first (less) blurred texture will be set as main texture of the ■SpokesSpin material when the wheel speed will be **minSpeed** and the last (most) blurred texture will be set as main texture of the ■SpokesSpin material when the wheel speed will be a little less than **maxSpeed**.*

3.4 Adaptive Occlusion

If the Adaptive Occlusion feature is enabled, another pass is made in order to occlude the inner part of the wheel when the angle of the camera relatively to the wheel is high.

The original blurred texture is replaced by one with extra alpha, so this cause the GPU create one more texture for each wheel and depending on the texture size this could be expensive.



The angle between the camera and the wheels depends on the orientation of the single wheel (left or right), so all the wheels on left side must have a `localScale.x` less than zero (the opposite value of the corresponding right wheel). Check the example scene to see how the wheels are positioned. It's easy to customize the way S.W.E. gets informations about the wheel orientation: just change the lines 362 and 390 of the `SpinEffect.cs` script.

4. Other uses

*S.W.E. is an effect for car games but it can work with other kind of games. For example, it can simulate a plane/helicopter propeller or a tourbillon mechanism. The **Plane** or **Tourbillon** example scenes are a good start for these particular applications of S.W.E.*

